**Objects & Prototypes**

Different ways of creating an object in javascript

**Object literal**

var user = {name:’Arun’, location:’United States’};

user.name will get you the name from the object

user.email = “[arun.gopan@marlabs.com](mailto:arun.gopan@marlabs.com)”; // adding new property in to an object

user.display\_details = function() {

return this.name+', '+this.location+', '+this.email;

}; // Adding a new function to an object

**Constructor function**

Allows to create multiple instance of an object which has the same structure.

function user(name, location) {

this.name = name;

this.loc = location;

}

var user1 = new user(‘Arun’, ‘United States’);

**this keyword**

Refers to an object, whatever object which executing the current bit of code. By default it the global object, in the web browser it is the window object.

In the above example the **this** refers to an new empty object, which is created using the **new** keyword and the new keyword sets the context of **this** to the new object.

If you try to call the **user()** without the new keyword. It returns undefined, because the **user()**  is not returning anything, previously it was the new keywords which was returning you the object.

Now the **this keyword** refers to the global object(whatever object which is executing the current bit of code), the window object. So if you try **window.loc** you should get the location value. (without the new keyword)

**Object.create()**

Above mentioned methods are just short hands for this. Both the cases this is what is happening the background.

var user = Object.create(Object.prototype, {

name : {

value : "Arun",

enumerable : true,

writable : true,

configurable : true

},

loc : {

value : "United states",

enumerable : true,

writable : true,

configurable : true

}

});

Object.prototype : object that becomes the prototype for the newly created object

**ECMA Script 6 Classes to create Objects**

Provides functionality for creating objects using a **class structure**

**Syntax :**

class user{

constructor(name, loc) {

this.name = name;

this.loc = loc;

}

display\_details() {

return 'Name = '+this.name+', Location = '+this.loc;

}

}

var user1 = new user('Arun', 'United States');

**Object properties**

**Property descriptor :** return all the properties of an object.

Object.getOwnPropertyDescriptor(user1, 'name')

Object.getOwnPropertyNames(user1) : list all the property names.

Object.keys : return all the property names.  
 var user = {

firstname : "Arungopan",

lastname : "Gopakumar"

};

Object.keys(user)

Object.values(user)

**Change property of an object :** use **Object.defineProperty()**

**Writable property**

**Object.defineProperty(user, 'name', {writable: false})** // makes the name property non writable.

If the name contains a another object, that can be still changed, for example.

name:{firstname:'Arun', 'lastname':'Gopan'}, still the firstname and lastname can be changed as we have only blocked the name from writing.

To completely block the name property use **Object.freeze(),** (prevents the object from changing the properties and editing values etc)

Object.freeze(user);

**Enumerable property**

Which allows to loop through the object.

If set to false it will not be available to loop through and even if the object is converted to JSON string using JSON.stringify function, it won't be available.

Object.defineProperty(user, 'name', {enumerable: false})

document.write(JSON.stringify(user));

**Configurable property**

If set to false :

Cannot change enumerable, configurable and cannot delete a property.

However you can still change the writable property.

**Getters and Setters**

Allows to specify the return value of a property and set the value of a property using a function.

Object.defineProperty(user, 'fullName', {  
 get: function() {  
 return this.name.firstname+', '+this.name.lastname;  
 },  
 set: function(value) {  
 var name\_split = value.split(' ');  
 this.name.firstname = name\_split[0];  
 this.name.lastname = name\_split[1];  
 }  
});  
  
user.fullName = "Arungopan Gopakumar";  
  
console.log(user.fullName);  
console.log(user.name.firstname);  
console.log(user.name.lastname);

**Prototypes**

Prototype is an object that exists in every function, Objects in javascript.

When an object is created in javascript it inherits all its methods and properties from Object.prototype (which is again an object).

**Function prototype :** Is the object instance which become the prototype for all the object created using this function as a constructor

var myFunc = function() {

return 'Hi';

}

console.log(myFunc.prototype);

Whenever a function is created a prototype is automatically created and attached.

**Object prototype :** Object instance from which the object is inherited.

var myObj = {name:"Arun", loc:"US"};

console.log(myObj.prototype); // returns undefined

console.log(myObj.\_\_proto\_\_);

Example to demonstrate function prototype and object \_\_proto\_\_ are the same

function user(name, location) {

this.name = name;

this.location = location;

}

var user1 = new user('Arun', 'US');

console.log(user.prototype);

console.log(user1.\_\_proto\_\_);

console.log(user.prototype===user1.\_\_proto\_\_)

Now if you add a new property to function prototype

user.prototype.age = 30; it gets automatically available to object created from it.

**Note :** If you check the user1 object, it won't have a property called age, only its \_\_proto\_\_ has.

If you change the age property of user1 by adding user1.age = 20, now is when it creates a new property age in user1 object.

Whenever we request a property of an object, javascript first checks whether the object has that property, if not it will check its prototype and returns it.

If we change the value of the age using **user.prototype.age** it gets automatically applied to all the instance derived from it. But what happens if we change the prototype to point to a completely new object.

**user.prototype = {age:40}**

var user2 = new user(‘Jacob’, ‘India’);

Now the user1 will have age as 30 where as the user2 will have the age as 40.

Which points out that the user1 object has prototype pointing to an instance of object in memory

**Multiple level of inheritance**

function user(name, loc) {

this.name = name;

this.loc = loc;

}

user.prototype.age = 30;

var user1 = new user('Arun', 'US');

console.log(user1.\_\_proto\_\_); // pointing to user prototype

console.log(user1.\_\_proto\_\_.\_\_proto\_\_); // pointing to object prototype

console.log(user1.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_); returns null

By default all objects in javascript inherit from object and object has not prototype.

**Lets create our own prototype chain**

function human() {

}

human.prototype.display = function() {

return 'I am human';

}

Now pointing the user to have prototype human

function user(name, loc) {

this.name = name;

this.loc = loc;

}

user.prototype = Object.create(human.prototype);

Now the user has the prototype human.

If you check

* user1 instanceof human
* user1 instanceof user

Both holds true, but the constructor function of user is not correct it will be pointing to human and we need to correct that as well by adding the code below.

user.prototype.constructor = user;

Now since the user has a prototype set to human, you can access the methods of human.

Now lets try to add a new property to **human()** and try to access it from **user1 instance.**

And you see that it returns undefined. That is because we haven't made the human available inside user. To do that we need to call human() inside user()

**human.call(this);**

**Note :** A different **this object** can be assigned when calling an existing function. this refers to the current object, the calling object. With **call**, you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

**human.apply(this)**

Yet another way of invoking function, it only differs from **call()** for the way by which the arguments are passed to the function. (arguments passed as an array where as in call, arguments passed as individual)

**Putting it all together.**

function human(text) {

this.abc = text || "custom";

}

human.prototype.display = function() {

return 'I am human';

}

function user(name, loc) {

human.call(this); // you are passing the **this** to the calling function (**this** refers to the current object, using call you can inherit all the properties of the calling method)

this.name = name;

this.loc = loc;

}

user.prototype = Object.create(human.prototype);

user.prototype.constructor = user;

var user1 = new user('Arun', 'US');

console.log(user1 instanceof human);

console.log(user1.abc);

**Closure**

Reference : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

A closure happens when you return a function a from inside a function and the inner function retains access to the scope.

**Example 1 :**

var closureAlert = function() {

var x = 0;

var alerter = function() {

alert(++x);

}

return alerter;

}

var closureFn = closureAlert();

var closureFn2 = closureAlert();

closureFn(); // alerts 1

closureFn(); // alerts 2

closureFn2(); // alerts 1

closureFn2(); // alerts 2

**Example 2 :**

var add = function(num) {

var num1 = num;

var addToNum1 = function(num2) {

return num1 + num2;

}

return addToNum1;

}

var add5 = add(5);

add5(7);

**Closure within a loop**

* function showVal(value) {  
   return function() {  
   console.log(value);  
   };  
  }  
    
  for(var i=0; i<5; i++) {  
   setTimeout(showVal(i), i\*1000);  
  }

**Closure objects**

You can even return object instead of a function directly and the same scope holds.

**Example :**

var closureAlert = function() {

var x = 0;

return {

count : function() { alert(++x); },

reset : function() { x=0; }

};

}

var closureFn = closureAlert();

closureFn.count();

closureFn.count();

closureFn.reset();

closureFn.count();

closureFn.count();  
  
**Private functions with closure**

var a = (function () {

var privatefunction = function () {

alert('hello');

}

return {

publicfunction : function () {

privatefunction();

}

}

})();

console.log(a.publicfunction());

**Ajax**

* Ajax stands for asynchronous JavaScript and XML
* It runs in the background
* Helps you to add update content in your web page without reloading it
* Allows to request / receive data from server even after the page has been fully loaded   
    
    
  var xhttp = new XMLHttpRequest();  
   xhttp.onreadystatechange = function () {  
   if (xhttp.readyState == 4 && xhttp.status == 200) {  
   document.getElementById("demo").innerHTML = xhttp.responseText;  
   }  
   };  
   // get method  
  // xhttp.open("GET", "test.php", true);  
  // xhttp.send();  
    
   //post method  
   xhttp.open("POST", "test.php");  
   var formData = new FormData();  
   formData.append('username', 'Arungopan');  
   formData.append('email', 'arun.gopan@marlabs.com');  
   xhttp.send(formData);

**Promise**

The Promise object is used for deferred or asynchronous computations. A Promise represents a value which may be available now, or in the future, or never.

var promise = new Promise(function(resolve, reject) {

//resolve('Hello, this is the response from resolve function.');

reject('Hello, this is the response from reject function.')

});

promise.then(function(data) {

console.log(data);

},

function(data) {

console.log(data);

});

In the above example both resolve and reject are methods which indicates whether the promise is resolved or rejected.

then(), has two function, first function for the handling resolved promise and second for rejected promise.

**bind()**

The bind() method creates a new function that, when called, has its this keyword set to the provided value.

Use .bind() when you want that function to later be called with a certain context, useful in events.

Use .call() or .apply() when you want to invoke the function immediately

**Example for bind :**

this.username = "Arungopan";

this.place = "New Jersey";

function user() {

return this.username+', '+this.place;

}

var obj = {

username : "Nandakumar",

place:"Piscataway"

}

var new\_fn = user.bind(obj);

document.write(new\_fn()+"<br />");

document.write(user()+"<br />");  
  
**Example 2 :**

var user = {

"username":"Arungopan",

"place":"Piscataway",

"clickFn": function() {

alert(this.username+", "+this.place);

jQuery(this).html('Helloo');

}

}

jQuery(document).on('click', '#btn', user.clickFn.bind(user));

**Example 3 :**

function addTwoNumbers(number2) {

this.number2 = number2;

return this.number1 + this.number2;

}

var addToFive = addTwoNumbers.bind({number1:5});

var addToTwenty = addTwoNumbers.bind({number1:20});

console.log(addToFive(10));

console.log(addToTwenty(10));

**Using bind with addEventListener**

* Track clicks (or to perform an action after a click) that might require us to store information in an object  
  + var logger = {  
     x: 0,  
     updateCount: function(){  
     this.x++;  
     console.log(this.x);  
     }  
    }
  + We might assign click handlers like this and subsequently call the updateCount() in our logger object
  + document.querySelector('button').addEventListener('click', function(){  
     logger.updateCount();  
    });
* Instead of using an anonymous function to call the logger function, the above example can be modified to use **bind()**
  + document.querySelector('button').addEventListener('click', logger.updateCount.bind(logger));